A Case Study in Capacity Planning for PEPA Models with the PEPA Eclipse Plug-in

Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 318 (2015) 69–89

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

A Case Study in Capacity Planning for PEPA Models with the PEPA Eclipse Plug-in

Christopher D. Williams and Allan Clark[1,2,3]

LFCS, School of Informatics
University of Edinburgh
Edinburgh, United Kingdom

Abstract

We report on the addition of *Capacity Planning* facilities to the PEPA Eclipse Plug-in, a software tool for analysing performance models written in the PEPA language. The PEPA language allows the compositional description of complex systems consisting of different kinds of processes. The capacity planning addition allows modellers to automatically search for the populations of processes that allows for an optimal trade-off between the performance of the system and the cost of acquiring or operating the components of the system under the modeller's control.

Keywords: PEPA, Capacity Planning, Optimisation, Performance Evaluation, Modelling

1 Introduction

In this paper we report on the capacity planning framework for Performance Evaluation Process Algebra (PEPA)[8] implemented in the PEPA Eclipse Plug-in[16]. PEPA is a language in which modellers can compositionally describe complex systems. Generally modellers define several different kinds of processes which interact with each other by sharing activities. Once the model is defined, it can be numerically evaluated via a suite of techniques to obtain performance metrics. If the model is accurate enough then these translate to, and provide insight to, the real system under investigation.

Typically a process is defined with several possible states. The performance metrics in the first instance are the long-term probabilities of a process being in each of its possible states. Where the model contains many copies of the same process, this is equivalent to asking the long-term populations of each state.

[1] Special thanks to Mirco Tribastone who contributed the case study

[2] Email: c.d.williamsed.ac.uk

[3] Email: a.d.clark@ed.ac.uk

http://dx.doi.org/10.1016/j.entcs.2015.10.020

Generally, the populations, will provide the modeller with utilisation information. For example in a *Server-Client* model, a server process may have a set of possible states, some of which correspond to a state in which the server is busy processing a particular kind of response and other states will correspond to a state in which the server is idle waiting for a client to make a request. Knowing the proportion of servers which are generally busy may tell the modeller if the service is over-provisioned, in that we have many servers which are idle because the number of servers is enough to satisfy the expected (and modelled) demand.

From the model and the long-term/steady-state population distributions we can derive performance measures which may more directly determine whether the service would be over or under provisioned. Two important measures are the throughput of particular actions or the time a particular component can expect to remain in a particular set of states. Again in a *Server-Client* setting, the throughput may tell us how many requests are responded to per unit of time, or it may tell us the rate at which requests must be dropped.

However the throughput of requests is commonly also dependent upon the rate at which requests are made, and hence may not be a reliable indicator of whether the system has enough performance for a given demand. For this we can calculate the expected time a given component is in a given set of states. Typically we would calculate the expected time a single client is in a state in which they have made their *request* and are waiting for that request to be responded to by the service. This gives us the response-time as observed by a typical consumer of the service.

Being able to predict the performance of a proposed service by modelling the service and calculating the response-time under a given client-load is clearly useful. However when designing the system we still have some flexibility around the number of components that we may deploy. In the simple case we may be able to deploy more or fewer servers. In a more complex environment there are different components that make up the service being offered. For example there may be web servers and database servers as well as an external authentication service.

When designing such a system, we would like to know what configuration of the system is optimal. So we wish to know what populations of server components meets some required performance standard. One can often meet a performance standard simply by deploying ever-increasing numbers of all server components. However, there is generally some cost associated with acquiring and/or operating each server and the costs may differ for different types of server. Hence we wish to find not only a configuration of the system that will satisfy the performance demands but also the cheapest such system configuration.

A modeller can always guess at a sufficiently low-cost configuration that might satisfy the performance demands and simply evaluate that configuration through their model. In previous work [13], an extension to the PEPA Eclipse Plug-in software is discussed. The extension implements an automatic search for the optimal configuration, ie. the extension implements automatic capacity planning for PEPA models.

In this paper we contribute a case-study demonstrating the value of the capacity

planning extension to the PEPA Eclipse Plug-in. We begin in the following section with background information detailing PEPA, associated performance measures and capacity planning in general. This is then followed by a in-depth description of the case study scenario and the associated PEPA model. The results obtained from running the software to obtain an optimal configuration of the server components in the case study are then discussed. We end with future work discussing how to make the software ever more general without sacrificing ease-of-use and the conclusion that the capacity planning extension is an important feature for modelling software.

2 Background

This section gives an overview of the PEPA modelling language and the necessity to provide capacity planning facilities. We first discuss PEPA, then describe how PEPA models can be evaluated to obtain basic quantitative information concerning how the model's component states evolve over time and their steady-state (or long-term) distributions. We then describe how more informative performance measures can be derived from this basic information. Crucially we are looking at the throughput of particular activities within the model as well as the expected time the model remains in a particular set of states. The latter is known as the average response-time in the specific case where the set of states represents a client waiting for service/response from a server.

2.1 PEPA

PEPA is a stochastically-timed process algebra where sequential components are defined using prefix and choice. PEPA models require these sequential components to cooperate on some activities, and hide others. A PEPA model typically consists of several sequential components, placed in cooperation. In the model:

$$P \underset{\mathcal{L}}{\bowtie} Q$$

The sequential components P and Q cooperate on the activities in the set $\mathcal{L}$. If activity α is in the set $\mathcal{L}$ then P and Q are required to cooperate on α. If activity β is not in $\mathcal{L}$ then either of P or Q, or both, may perform this activity independently. When $\mathcal{L}$ is empty we write $P \parallel Q$ instead of $P \underset{\emptyset}{\bowtie} Q$. We also allow the special cooperation $P \underset{*}{\bowtie} Q$ to be a synonym for $P \underset{\mathcal{L}}{\bowtie} Q$ where $\mathcal{L}$ is the set of activities performed by both P and Q.

Rates are associated with activities performed by each component. The symbol $\top$ is used to indicate that the component will passively cooperate with another on this activity. In this case the passive component may enable or restrict the activity from being performed by the cooperating component but the rate when enabled is determined by the actively cooperating component. The component $(a, r).P$ performs the activity a at rate r whenever it is not blocked by a cooperating component and becomes the process P. The component $(a, \top).Q$ passively synchronises on a and becomes process Q.

In PEPA models we often work with *arrays* of components. We use arrays to

represent workload (such as a number of independent clients) or resources (such as a number of independent servers). We write $P[5]$ to denote five copies of the component P which do not cooperate and $P[5][\alpha]$ to denote five copies of the component P which cooperate on the activity α. That is, $P[5]$ is an abbreviation for $P \parallel P \parallel P \parallel P \parallel P$ and $P[5][\mathcal{L}]$ is an abbreviation for

$$P \underset{\mathcal{L}}{\bowtie} P \underset{\mathcal{L}}{\bowtie} P \underset{\mathcal{L}}{\bowtie} P \underset{\mathcal{L}}{\bowtie} P.$$

The PEPA language is formally defined in [8]. Applications of the language are described in [7,11,10].

2.1.1 Evaluating the Model

PEPA is used to calculate performance measures. Traditionally this has been achieved by translating the PEPA model into its underlying continuous-time Markov chain. However for models with large numbers of components the state-space of the model is too large to traverse and other techniques have been utilised. We have used stochastic simulation and translation into Ordinary Differential Equations (ODEs) [9,17].

In this work we have utilised the translation into ODEs. The techniques described are generalisable to any method of obtaining results from a PEPA model, however if the search space is large we may evaluate many instantiations of the model, meaning that whichever method is used should be fast for all the candidate configurations of the model.

When the model is translated into a set of ODEs these can be numerically evaluated to provide a time series which describes the population levels of the states of each component kind over time. For some measures we are not interested in the evolution of the component state populations but rather the long-term proportion of the components in each state, known as the steady-state. To obtain these the ODEs can be numerically evaluated for increasing time until the populations are stable. This technique requires that your system does not exhibit oscillating behaviour.

2.1.2 Performance Measures

Once we have the long-term component populations we can calculate the expected performance of the system. The two most common performance measures that we are interested in are the throughput of particular actions or the average duration of a particular state, or set of states. For example when considering some kind of service we may wish to measure the throughput of responses made or the average response-time. The average response-time here would be the expected delay between a particular client making a request and that same client receiving a response to the earlier request.

Typically the average response-time is appropriate because it is not a measure that is penalised when the service is over-provisioned. When the service is over-provisioned the performance may be very high, but the throughput of responses can only be as high as the throughput of requests made by the users of the service. Response-time on the other hand can still be, and is likely to be, low, even when

the rate of requests is low. In the case study presented in this work we focus on response-time.

Response-time can be calculated with an application of Little's Law[14]. Little's law states that the long-term average number of customers in a stable system L is equal to the long-term average arrival rate λ multiplied by the average time a customer spends in the system W. Hence, $L = \lambda W$, re-arranging for W we have that $W = L/\lambda$. In our case W is the response-time we seek, and L is the long-term population of clients in states between their request and response activities whilst λ is long-term throughput of requests made.

When considering systems with many consumers we wish to evaluate the performance of the system as observed by a typical consumer. In other words we must be careful to measure the expected time between a request made by a particular client and the response received by that same client. Rather than the expected time between any request and the next response to any client. To achieve this we use a simple technique of *tagging* a single client, similar to the technique described in [5]. Tagging and specifying the states to be considered as part of the response-time, or the specific actions considered part of the measured throughput are discussed in [2] which introduces extended stochastic probes as a means for performance query specification. Automatically modifying the model to suit the performance query is discussed in [3].

Often a modeller must be careful to evaluate both response-time and throughput. Since a low throughput of requests may results in a low average-response time, but only because, for some reason, the demand is low. Similarly with no bound on the number of clients in a waiting state may mean that the throughput of responses is high, but that clients are waiting a long time for their requests to be satisfied.

2.2 Capacity Planning

Complex systems are commonly modelled to provide insight. Either this insight is used to aid the design of the system before it is built or it is used to understand a system already in operation. A complex system may have many components such as different kinds of servers. One aspect of the design of a system is reasoning about the most appropriate configuration, that is the numbers of different kinds of components. Modelling of a system can allow one to speculate about the most appropriate configuration and compare candidate configurations without physically implementing them.

Once efficient comparison of multiple candidate configurations is possible, it makes sense to perform a search for the best or most appropriate configuration. Several techniques exist for searching a space of candidate parameter configurations. Capacity planning has the unique property that usually the parameters in question are all integers since they represent a physical configuration of a system in terms of the populations of component types.

When considering configurations there are generally some trade-offs to consider. Usually some or all of the components over which the designer has control of their populations, have some cost associated with obtaining and running those compo-

nents. For example a web-service must spend money to obtain and run the physical (or virtual) servers which host the web-service. However the designer will also wish to ensure that the service gives enough performance such that, for example, the response-time is sufficiently low.

We already know how to obtain performance measures from a PEPA model such as response-time as explained above. Suppose we have a model with only one kind of server and a desired average response time from a given level of service demand or number of users. In this simple scenario it is straightforward to find the optimal configuration. We can simply evaluate the response-time when just a single server is allocated. If this response-time is low enough then this configuration can be reported as the most appropriate configuration knowing that all other configurations will cost more. If not then the model can be modified such that there are two servers and re-evaluate the response-time. In this way a simple brute-force search for the lowest number of servers which satisfies the response-time can be conducted.

In this simple case the first configuration found will be the most appropriate since we know that all other configurations we have tried do not satisfy the desired response-time and all configurations not yet tried will cost more. An obvious improvement would be a binary search. Both techniques would be performing a full search of the configuration space and guaranteed to find the best configuration.

However complex systems are often not as simple. There may be several kinds of servers each with a different cost to obtain and/or operate. Such a linear search for the best candidate solution can still be done if the candidate configurations can be ordered in terms of their operating cost. This may become prohibitively expensive to perform when the number of candidate configurations increases rapidly. Binary search may help in this scenario, but only if the candidate configurations are trivial to order and index in terms of cost.

Furthermore it may also be that the modeller does not have a strict performance threshold to achieve, but simply wants to trade-off good performance against the cost of operation. In this case the modeller might give a notional *cost* to each unit of response-time. Hence the *cost* of a candidate configuration is a combination of the cost of obtaining and operating the components *and* the predicted response-time achievable under that configuration.

In such a scenario we cannot do a linear search and stop at the first candidate configuration which satisfies the performance constraints because there are no strict performance constraints and a better trade-off may exist. One must search over the terrain of the entire search space. In this kind of search it may still be possible to perform a brute-force search and simply evaluate all possible configurations. This is only possible if the number of plausible configurations is low.

When brute-force search is not possible, search techniques exist which avoid evaluating all possible configurations. The work described here utilises such techniques specifically for PEPA models with associated performance measures. In essence then capacity planning is a *search* for the optimal configuration. Search heuristics make it possible to perform a search over a very large space of possible configurations without evaluating all possible configurations. This means that the absolute

best configuration may not be found. However, even in such cases it is worth performing the search automatically. Such techniques often fall under the category of *evolutionary computing* of which there is a large literature base, for example [15,4,1].

2.2.1 Cost Functions

The evaluation of a particular configuration involves assessing how appropriately the configuration balances performance and operating cost. These two measures are combined into an overall *cost* of a candidate configuration. Hence, where, $cost_{pm}$ is the cost associated with the performance measure, $cost_{pop}$ is the cost associated with the candidate configuration populations and w_{pm} and w_p are weights, the *cost function* has the general form:

$$cost = (w_{pm} \times cost_{pm}) + (w_p \times cost_{pop})$$

However not all populations are weighted equally. One possible task is deciding how many of each different kind of server to deploy and it may well be that the different kinds of servers do not cost the same to acquire or operate. For N component kinds, P_i is the population of component kind i and C_i is the cost associated with a single component of kind i. Hence our populations component of the cost function becomes:

$$cost_{pop} = \Sigma_1^N C_i \times P_i$$

Recall that our performance measure may be associated with the throughput of an action or the average response-time. Additionally in either case we may desire either a high or low value. Hence the cost function must be capable of penalising both a high or a low value for a performance measure. The simple solution is to set a target value for the performance measure and calculate the difference from this value. The modeller sets the target and the direction, so for a performance measure which we wish to search for as low a value as possible we have:

$$cost_{pm} = measured - target$$

Similarly for a performance measure for where we are searching for as high a value as possible we have:

$$cost_{pm} = target - measured$$

Note that in both cases this value may become negative. That is perfectly acceptable and there is no reason to avoid negative costs. The search engine will simply search for the configuration which gives the lowest cost, whether that lowest value is negative or not.

However this simple measure assumes that the modeller would penalise performance linearly. Consider the case of measuring average response-time. The modeller may be interested in keeping the average below that which is noticeable by users and hence may be very keen to discard configurations which evaluate to a response-time of 1.0 time units or greater over those which evaluate to a response-time of less than 1.0 time units. However, the modeller may be less concerned about distinguishing between two configurations that evaluate to response-times of 0.1 and 0.2 time units. Preferring instead to distinguish those two configurations more according to the cost of the populations.

Providing non-linear cost functions adds significantly to the complexity of the

user-interface provided for the modeller to specify their cost function. Hence we have approximated non-linear cost functions by utilising a penalty for missing the target. Hence our cost function for a performance measure for which we wish to search for the lowest possible value becomes:

$$cost_{pm} = (target - measured) + (H(target - measured) \times penalty)$$

Where H is the Heaviside function which returns zero when given a negative argument and one otherwise. This corresponds to a situation in which you may wish to adhere to a given service level agreement. For example the service level agreement may state that the average response-time observed by users is no more than 0.5 time units. Hence we can heavily penalise all configurations which result in the model predicting an average response-time of more than 0.5 time units. This means that for configurations which result in a response-time better than 0.5 time units the search will still prefer better configurations, but that the weights on the performance and population costs can be set sensibly.

For configurations in which the average response-time computed is worse than the target then the penalty is applied. This allows the search to reject such configurations regardless of the population cost.

Recall that capacity planning is a search for the optimal configuration. Avoiding a brute-force search of all configurations is important because the search space of configurations is often infeasibly large. Search heuristics can avoid an exhaustive search by finding good areas of the search space. To enable this it is important that our cost function enables the search to be directed towards good areas in the search space. For this reason there is still a gradient on the performance cost when the target is not reached. That is, a configuration that misses the target response-time by a little, still evalutes to a lower (performance) cost than a configuration which misses the response-time target by a larger amount. This helps the search to move towards configurations in which the target will be met, rather than simply rejecting those that fail to meet the response-time target.

The weights of the overall cost function w_{pm} and w_{pop} allow the user to adjust the importance of reducing the cost associated with the performance measure against the cost associated with the populations of the components. The task that the user has is to set these weights such that neither component of the cost dominates the overall cost.

Unfortunately it is impossible for us to set a useful default here since we cannot know in advance how the populations are costed. In addition the unit used for the rates in the model is undefined. Hence determining how costly each unit of, for example, response-time is, is a task that is necessarily left for the user. To see this, consider that a model which used seconds as the unit can have all rates in the model multiplied by 1000. The steady-state probabilities will not change, but any response-time measure we calculate will now be in the units of milliseconds rather than seconds. This new model is just as valid, but clearly there would be a much smaller real cost associated with a one time unit rise in average response-time. Hence the weights used in this model would need to reflect that.

Strictly speaking the weights are unnecessary since the user could always adjust

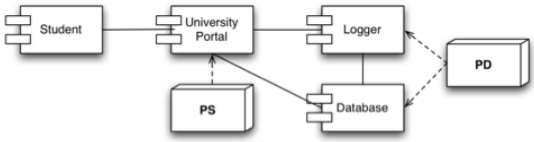

Fig. 1. Deployment diagram of the e-University case study. Solid connectors between components indicate request/reply communication. Dashed lines denote the deployment of services onto processors.

the weights on the populations, but we include them as a convenience for the user. Typically the user will not run a single capacity planning search, but will run several searches modifying their search parameters accordingly.

We have given a brief description of the cost function considerations in this section. Cost functions can become very complicated. There is a natural tension between providing clear and usable software whilst also covering as many use cases as possible. Ultimately to be entirely generic would require that we allow the user to calculate their own cost function in some full evaluation environment such as a general purpose programming language. This currently remains future work but for now we hope to have provided enough flexibility for common use cases without adding significant bloat and complication to the software and its associated user interface.

3 Case Study

Our example scenario concerns a previously studied [12] scenario which formed part of a case study of the SENSORIA project [6, Chapter 2]. It concerns a hypothetical European-wide virtual university in which students study remotely. The part of the case study considered in the above work and in this work concerns the course selection phase where students already matriculated to the university must enrol in specific courses. Although the students only enrol in a few courses per year they all do this at the same time, so it is important that sufficient provision is provided to maintain a responsive service.

The case study is comprised of a number of scenarios; here the scenario of interest is the Course Selection scenario, where students obtain information about the courses available at their education establishment and may enrol in those for which specific requirements are satisfied. Although the overall application is intended to be service-oriented, the scenario investigated here is such that the kinds of services available in the system do not to change over the time frame captured by this model. This reflects the fact that a university's course organisation is likely to be fixed before it is offered to students. Furthermore, minor changes are likely not to affect the system's behaviour significantly. The model will not consider other services which may be deployed in an actual application (e.g. authentication services) because their impact on performance is assumed to be negligible. The scenario also considers a constant population of students to capture a real-world situation where the university's matriculation process is likely to be completed before the application may be accessed.

3.1 The Model

The current authors are indebted to the authors of the above mentioned study [12] for allowing us to include their description of the model here.

The access point to the system is the University Portal, a front-end layer which presents the available services in a coherent way, for example by means of a web interface. There are four services in this model:

Course Browsing allows the user to navigate through the University's course offerings;

Course Selection allows the user to submit a tentative course plan which will be validated against the University's requirements and the student's curriculum;

Student Confirmation will force the student to check relevant personal details;

Course Registration will confirm the student's selection.

These components make use of an infrastructural *Database* service, which in turn maintains an event log through a separated *Logger* service.

The modelling paradigm adopted here captures the behaviour of a typical multi-threaded multi-processor environment used for the deployment and the execution of the application. The *University Portal* instantiates a pool of threads, each thread dealing with a request from a student for one of the services offered. During the processing of the request the thread cannot be acquired by further incoming requests, but when the request is fulfilled the thread clears its current state and becomes available to be acquired again. Analogous multi-threaded behaviour will be given to the *Database* and *Logger* components.

Performance issues may arise from the contention of a limited number of threads by a potentially large population of students. If at some time point all threads are busy, further requests must queue, provoking delays and capacity saturation. This model also proposes another level of contention by explicitly modelling the processors on which the threads execute. Here, delays may occur when many threads try to acquire a limited number of processors available. Furthermore, this may be worsened by running several multi-threaded services on the same multi-processor system, as will be the case in the deployment scenario considered in this model: *University Portal* will run exclusively on multi-processor *PS*, whereas *Logger* and *Database* will share multi-processor *PD* (see Figure 1).

3.1.1 General Modelling Patterns

Processing a request involves some computation on the processor on which the service is deployed. Such a computation in the PEPA model is associated with an activity *(type, rate)*, where *type* uniquely identifies the activity and *rate* denotes the average execution demand on the processor (i.e. $1/rate$ time units). A single processing unit may be modelled using a two-state sequential component. One state enables an *acq* activity to acquire exclusive access to the resource, while the other state enables all the activities deployed on the processor. Letting n be the number of distinct activities, the following pattern is used for a processor:

$$\begin{aligned} Processor_1 &= (acq, r_{acq}).Processor_2 \\ Processor_2 &= (type_1, r_1).Processor_1 \\ &\quad + (type_2, r_2).Processor_1 \\ &\quad + \ldots \\ &\quad + (type_n, r_n).Processor_1 \end{aligned} \tag{1}$$

Communication in this model is synchronous and is modelled by a sequence of two activities in the form $(req_{from,to}, rreq).(reply_{from,to}, rrep)$ where the subscript from denotes the service from which the request originates and to indicates the service required. A recurring situation is a form of blocking experienced by the service invoking an external request. Let A and B model two distinct interacting services, for example,

$$\begin{aligned} A &= (req_{A,B}, r_{reqA}).(reply_{A,B}, r_{repA}).A \\ B &= (req_{A,B}, r_{reqB}).(execute, r).(reply_{A,B}, r_{repB}).B \end{aligned} \tag{2}$$

The communication between A and B will be expressed by means of the cooperation operator $A \underset{L}{\bowtie} B$ where, $L = \{req_{A,B}, reply_{A,B}\}$.

According to the operational semantics, A and B may initially progress by executing $req_{A,B}$, subsequently behaving as the process $(reply_{A,B}, r_{repA}).A \underset{L}{\bowtie} (execute, r).(reply_{A,B}, r_{repB}).B$.

Now, although the left-hand side of the cooperation enables $reply_{A,B}$, the activity is not offered by the right-hand side, thus making the left-hand side effectively blocked until *execute* terminates (i.e., after an average duration of $1/r$ time units). These basic modelling patterns will be used extensively in this case study, as discussed next.

3.1.2 University Portal

A single thread of execution for the application layer University *Portal* is implemented as a sequential component which initially accepts requests for any of the services provided:

$$\begin{aligned} Portal &= (req_{student,browse}, v).Browse \\ &\quad + (req_{student,select}, v).Select \\ &\quad + (req_{student,confirm}, v).Confirm \\ &\quad + (req_{student,register}, v).Register \end{aligned} \tag{3}$$

The rate v will be used throughout this model in all the *request*/*reply* activities. In the following, the action type acq_{ps} is used to obtain exclusive access to processor PS .

Course Browsing is implemented as a service which maintains an internal cache. When a request is to be processed, the cache query takes $1/r_{cache}$ time units on average, and is successful with probability 0.95, after which the retrieved data is

processed at rate r_{int}. Upon a cache miss, the information is retrieved by the *Database* service, and is subsequently processed at rate r_{ext}:

$$\begin{aligned} Browse &= (acq_{ps}, v).Cache \\ Cache &= (cache, 0.95 \times r_{cache}).Internal \\ &\quad + (cache, 0.05 \times r_{cache}).External \\ Internal &= (acq_{ps}, v).(internal, r_{int}).BrowseRep \\ External &= (req_{external,read}, v).(reply_{external,read}, v). \\ &\quad (acq_{ps}, v).(external, r_{ext}).BrowseRep \\ BrowseRep &= (reply_{student,browse}, v).Portal \end{aligned} \tag{4}$$

Course Selection comprises four basic activities. An initial set-up task initialises the necessary data required for further processing ($rater_{prep}$). Then, two activities are executed in parallel, and are concerned with validating the selection against the university requirements ($rater_{uni}$) and the student's curriculum ($rater_{curr}$), respectively. Finally, the outcome of this validation is prepared to be shown to the student ($rater_{disp}$). The relative ordering of execution is maintained by considering three distinct sequential components. The first component prepares the data, then forks the two validating processes, waits for their completion, and finally displays the results:

$$\begin{aligned} Select &= (acq_{ps}, v).(prepare, r_{prep}).ForkPrepare \\ ForkPrepare &= (fork, v).JoinPrepare \\ JoinPrepare &= (join, v).Display \\ Display &= (acq_{ps}, v).(display, r_{disp}).SelectRep \\ SelectRep &= (reply_{student,select}, v).Portal \end{aligned} \tag{5}$$

The two validating processes are guarded by the fork/join barrier as follows:

$$\begin{aligned} ValUni &= (fork, v).(acq_{ps}, v).(validate_{uni}, r_{uni}).(join, v).ValUni \\ ValCur &= (fork, v).(acq_{ps}, v).(validate_{cur}, r_{cur}).(join, v).ValCur \end{aligned} \tag{6}$$

These components will be arranged as follows in order to obtain a three-way synchronisation:

$$Select \underset{fork,join}{\bowtie} ValUni \underset{fork,join}{\bowtie} ValCur \tag{7}$$

Student Confirmation is represented in the PEPA model as an activity performed at rate r_{con}. The service uses *Logger* to register the event:

$$\begin{aligned} Confirm &= (acq_{ps}, v).(confirm, r_con).LogStudent \\ LogStudent &= (req_{confirm,log}, v).(reply_{confirm,log}, v).ReplyConfirm \\ ReplyConfirm &= (reply_{student,confirm}, v).Portal \end{aligned} \tag{8}$$

Finally, *Course Registration* performs some local computation, at rate r_{reg}, and then contacts *Database* to store the information:

$$\begin{aligned} Register &= (acq_{ps}, v).(register, r_{reg}).Store \\ Store &= (req_{register,write}, v).(reply_{register,write}, v).ReplyRegister \\ ReplyRegister &= (reply_{student,register}, v).Portal \end{aligned} \tag{9}$$

The general pattern 1 is applied to processor PS as follows:

$$\begin{aligned} PS_1 &= (acq_{ps}, v).PS_2 \\ PS_2 &= (cache, r_cache).PS_1 + (internal, r_int).PS_1 \\ &\quad + (external, r_{ext}).PS_1 + (prepare, r_{prep}).PS_1 \\ &\quad + (display, r_{disp}).PS_1 + (validate_{uni}, r_{uni}).PS_1 \\ &\quad + (validate_{cur}, r_{cur}).PS_1 + (confirm, r_{con}).PS_1 \\ &\quad + (register, r_{reg}).PS_1 \end{aligned} \tag{10}$$

3.1.3 Database

This service exposes two functions for reading and writing data. Reading is a purely local computation, whereas writing additionally uses the Logger service. In this model, *Database* is only accessed by the university portal in states *External* and *Store* in equations 4 and 9, respectively. Let PD denote the processor on which *Database* is deployed, acquired through action acq_{pd} . Similarly to University *Portal*, a single thread of execution for *Database* is:

$$\begin{aligned} Database &= (req_{external,read}, v).Read + (req_{register,write}, v).Write \\ Read &= (acq_{pd}, v).(read, r_{read}).ReadReply \\ ReadReply &= (reply_{external,read}, v).Database \\ Write &= (acq_{pd}, v).(write, r_{write}).LogWrite \\ LogWrite &= (req_{database,log}, v).(reply_{database,log}, v).WriteReply \\ WriteReply &= (reply_{register,write}, v).Database \end{aligned} \tag{11}$$

3.1.4 Logger

This service accepts requests from *Student Confirmation* and *Database*, as described in equations 8 and 11, respectively. It is deployed on the same processor as *Database*, i.e., processor PD. Thus, one thread execution may be modelled as follows:

$$\begin{aligned} Logger &= (req_{confirm,log}, v).LogConfirm + (req_{database,log}, v).LogDatabase \\ LogConfirm &= (acq_{pd}, v).(log_{conf}, r_{lgc}).ReplyConfirm \\ ReplyConfirm &= (reply_{confirm,log}, v).Logger \\ LogDatabase &= (acq_{pd}, v).(log_{db}, r_{lgd}).ReplyDatabase \\ ReplyDatabase &= (reply_{database,log}, v).Logger \end{aligned}$$

(12)

Taking together 11 and 12 it is possible to write the sequential component that models the processor PD:

$$\begin{aligned} PD_1 &= (acq_{pd}, v).PD_2 \\ PD_2 &= (read, r_{read}).PD_1 + (write, r_{write}).PD_1 \\ &\quad + (log_{conf}, r_{lgc}).PD_1 + (log_{db}, r_{lgd}).PD_1 \end{aligned}$$

(13)

3.1.5 Student Workload

A student is modelled as a sequential component which interacts with the university portal and accesses all of the services available. The behaviour is cyclic and the student interposes some think time between successive requests. This results in a closed-workload type of behaviour which is typical of many performance studies:

$$\begin{aligned} StdThink &= (think, r_{think}).StdBrowse \\ StdBrowse &= (req_{student,browse}, v).(reply_{student,browse}, v).StdSelect \\ StdSelect &= (req_{student,select}, v).(reply_{student,select}, v).StdConfirm \\ StdConfirm &= (req_{student,confirm}, v).(reply_{student,confirm}, v).StdRegister \\ StdRegister &= (req_{student,register}, v).(reply_{student,register}, v).StdThink \end{aligned}$$

(14)

3.1.6 System Equation

The multiplicity of threads and processors is captured in the system equation, in which all the sequential components illustrated above are composed with suitable cooperation operators to enforce synchronisation between shared actions. The complete system equation for this model is:

$$\begin{array}{c} StdThink[N_S] \\ \bowtie_{*} \\ \Big((Portal[N_P] \underset{M_1}{\bowtie} ValUni[N_P] \underset{M1}{\bowtie} ValCur[N_P]) \\ \bowtie_{M_2} \\ Database[N_D] \underset{M_3}{\bowtie} Logger[N_L] \Big) \\ \bowtie_{*} \\ (PS1[N_{PS}] \underset{\emptyset}{\bowtie} PD1[N_{PD}]) \end{array}$$

where:

$$\begin{aligned} M_1 &= \{fork, join\} \\ M_2 &= \{req_{external,read}, reply_{external,read}, req_{register,write}, reply_{register,write}\} \\ M_3 &= \{req_{confirm,log}, reply_{confirm,log}, req_{database,log}, reply_{database,log}\} \end{aligned}$$

It is worth pointing out that the separate validating threads ValUni and ValCur inherit the multiplicity levels of the thread Portal which spawns them.

3.2 Performance Measure

Given this model we wish to measure and optimise for the performance observed by a typical student. We are therefore interested in calculating the average response-time for student requests. From the definitions in 14 the students in the $StdThink$ state are not attempting to make use of the system. We therefore calculate the average time it takes from the moment a student actively uses the system by moving into the $StdBrowse$ state until the student returns to the $StdThink$ state.

For the default configuration of the model we obtain the results 15.248 time units for the average response-time. We now wish to optimise the configuration of the system to obtain satisfactory performance whilst spending as little as possible on the components.

3.3 Capacity Planning

The populations that a designer of the hypothetical e-university service may be able to control are those of the components: $Database$, $Logger$, PD, $Portal$, PS, $ValCur$ and $ValUni$, we do not expect the service to be able to control the average number of students accessing the service simultaneously. Therefore the modeller assumes some fixed level of demand by fixing the initial population of $StdThink$, in this case to 600.

We are interesting in optimising for the response-time performance measure described above. In addition we would like to keep the cost of the system as low as possible.

We have limited data to allow us to obtain realistic rates for some of the activities in the model. Since the meaning of a unit of time in a PEPA model is unspecified we need only be concerned that the rates are of realistic magnitudes relative to each other.

In addition, as discussed in Section 2.2 we are able to place a threshold average response-time above which there is a heavy cost function penalty, to approximate a non-linear performance measure cost. In this study we somewhat arbitrarily specify the threshold to be 15, but this is no more arbitrary than the unspecified unit of time used in the PEPA model itself. The 15 in question comes from the fact that the hand-optimised version of the model for 600 users reported in [12] (mentioned above as the source of our case study's PEPA model), is 15.248.

Without specialist knowledge it is difficult to weigh the importance of the reducing the average response-time against the importance of reducing the cost of the system. So we have also somewhat arbitrarily set the cost function weights w_{pm} (the weight of the performance measure component) and w_{pop} (the weight of the populations components).

Hence we have two arbitrary pieces of information included in our cost function, that is the penalty threshold for the performance measure and the ratio of weight-

Optimisation	N_D	N_L	N_{PD}	N_P	N_{PS}	N_{VC}	N_{VU}	Total	Avg Res
Hand	80	80	40	80	40	80	80	480	15.248
Search	53	26	34	103	60	94	45	415	5.999
Brute	50	23	31	101	60	91	42	398	7.307

Table 1
Optimal configurations found for three search techniques. The first is manual optimisation, the second is our heuristic-based search and the last is a brute-force search around in a limited region around the optimal configuration found by the heuristic-based search. Both the heuristic and brute-force search find configurations that have a lower total population *and* a lower average response-time. The heuristic-based search finds the lowest average response time of the three whilst the brute-force search finds a lower total population.

ings for the performance measure and population components of the cost function. However both such pieces of information would be available to a real-world modeller utilising our capacity planning extension.

3.4 Results

The previously mentioned work which introduced our case study [12] utilised the ODE response-time evaluation to find a configuration of the model with a low response-time for the case in which there are 600 students. Table 1 shows the populations for configurations found, using three search methods. The first is the hand optimised configuration reported in the above referenced work, the second is our heuristic-based search and the last is for a brute force search, in which the search space was limited to a small area around the optimal configuration found by the heuristic based search.

In the hand-optimised case for the original publication the authors held three of the server populations to be equal to each other but not fixed. So in their case $N_{VC} = N_{VU} = N_P$, this was not a restriction that we imposed on our capacity planning search. Such a restriction is indeed easily imposed, but we wished to allow the search as much flexibility as possible.

Our software has found a model that has a significantly lower average response-time, with a response-time of 5.999 compared to 15.248. This would not be impressive if the cost of the server configuration were not cheaper. The population of every server component kind cannot be lower than the hand-optimised version otherwise we would have at best the same average response-time. However, in our case we obtain a model that has a total population of server components that is significantly less than that of the hand-optimised version. A total server component population of 415 against 480 for the hand-optimised version.

Half of the server components in the hand-optimised version has a lower population than the configuration found by our automatic search. These are, the number of *Portal* components N_P, the number of PS components N_{PS} and the number of VC components N_{VC}. In the case of the *Portal* component there are more than 25% more of them, for PS it is 50%, and lastly VC more than 15%.

As we have stated it is difficult to provide realistic costs for each of the server

components in such a hypothetical scenario. However our automatically optimised model is a significant improvement on the hand-optimised version unless *Portal* components or the multiprocessor systems that they run on are significantly more expensive than, for example, the multi-processors which execute the *Database* and *Logger* components.

We can assert that given our cost values for each of the components, the automatic search was able to find a configuration that had a significantly improved average response-time whilst simultaneously reducing the total cost of the server components.

The entire search took 9499 seconds, or under 160 minutes. In doing so it solved 1694 models. We can say that each model therefore took approximately 5.1 seconds to solve on average. Each model may take a different time to solve because the rates affect how quickly the model is solved. In addition there is some time spent performing the search algorithm logic, but this will serve us as an approximation. All of the computations described here were performed on a standard desktop computer. In addition it is the relative, rather than absolute times that we are mostly concerned with.

3.5 Brute Force Comparison

As described above the alternative to performing a stochastic search over the space of potential configurations is to perform a brute-force search evaluating all possible configurations.

The time taken to solve the set of ODEs generated from the PEPA model depends on the configuration, but is generally comparable across the configurations. As described above the capacity planning search is not instantaneous, but took around 160 minutes. The naïve approach to a brute-force search would evaluate all possible configurations within our initial constraints. This would have meant solving 671088640000000 distinct possible configurations and taken approximately 106400405 years.

A modeller could of course be a little more clever about the ranges set on population levels to reduce the search space. Whenever one does this there is a trade-off as you are trading-off the possibility that a better solution exists outside your narrower ranges against the advantage of your search performing faster.

However, the best solution had a highest population of 103 and a lowest population of 26. Even if we set all ranges to be from this lowest value 26 to the highest value 103, which would require insight into the search space that the modeller does not have, then the search space still has $(103-26)^7 = 16048523266853$ possible candidate configurations. Hence searching the entire space with a brute-force approach will still take 80242616334265 seconds or approximately 42407 years.

However, one could use the driven search to provide a suitable search area in which to perform an exhaustive search. To perform our brute-force search in a reasonable amount of time we set the ranges for each configurable population to a range around the value that we have found from the capacity planning search. To further reduce this we held the number of PS components constant at 60. As a

result our brute-force search had a more manageable number of configurations to solve: $46656 = (47-42)*(96-91)*(55-50)*(105-100)*(36-31)*(28-23)$. This too approximately 7.2 hours to solve. This best solution being shown in the table in Section 3.4.

3.5.1 Search Space

Figure 2 gives some idea of the search space of configurations. Each graph pair of graphs concerns one configurable component (in the interests of brevity we have included only two representative components, *Portal* and *Logger*). The left graph of each pair displays the results from the driven capacity planning search and the right displays the results from the brute-force search.

Each plotted dot represents a candidate configuration, the x-axis position determines the population of the candidate configuration for the particular server component kind depicted in that specific graph. The y-axis position determines the value of the cost function for that configuration. Recall that the cost function considers both the populations of all the configurable server components and the resulting average response-time.

The x-axis range on the brute-force search results are much narrower, because the brute-force search was centred on a narrow range around each optimal configuration found by the capacity planning search. This is because it is infeasible to do an exhaustive search for larger ranges.

The capacity planning graphs exhibit a lower left corner slope. This indicates that for each of these components there is a lower-bound on the population such that populations below this result in too high an average response-time, regardless of the rest of the configuration.

Each graph additionally demonstrates that one cannot optimise for each component kind independently. For each population of each component kind a wide range of costs are possible. Hence one must optimise for *all* of the configurable component populations simultaneously, because the population of one affects both the sensitivity and optimal value of another.

4 Future Work

Although we think that the user has been given much flexibility in the configuration of their cost function we realise that there are surely scenarios which call for some cost function that cannot be expressed using our configuration interface. A more general solution would be to allow the modeller to express their own cost function in a general purpose programming language such as Java which is used in our implementation.

To provide this, some interface to the results and the model parameters would be required. This would also place something of a burden on the modeller so we would be keen to retain a simple gui-based configuration scheme that may be used as a first exploration of the configuration space, and/or by novice users.

Recall that our practice of having the user specify a target performance measure

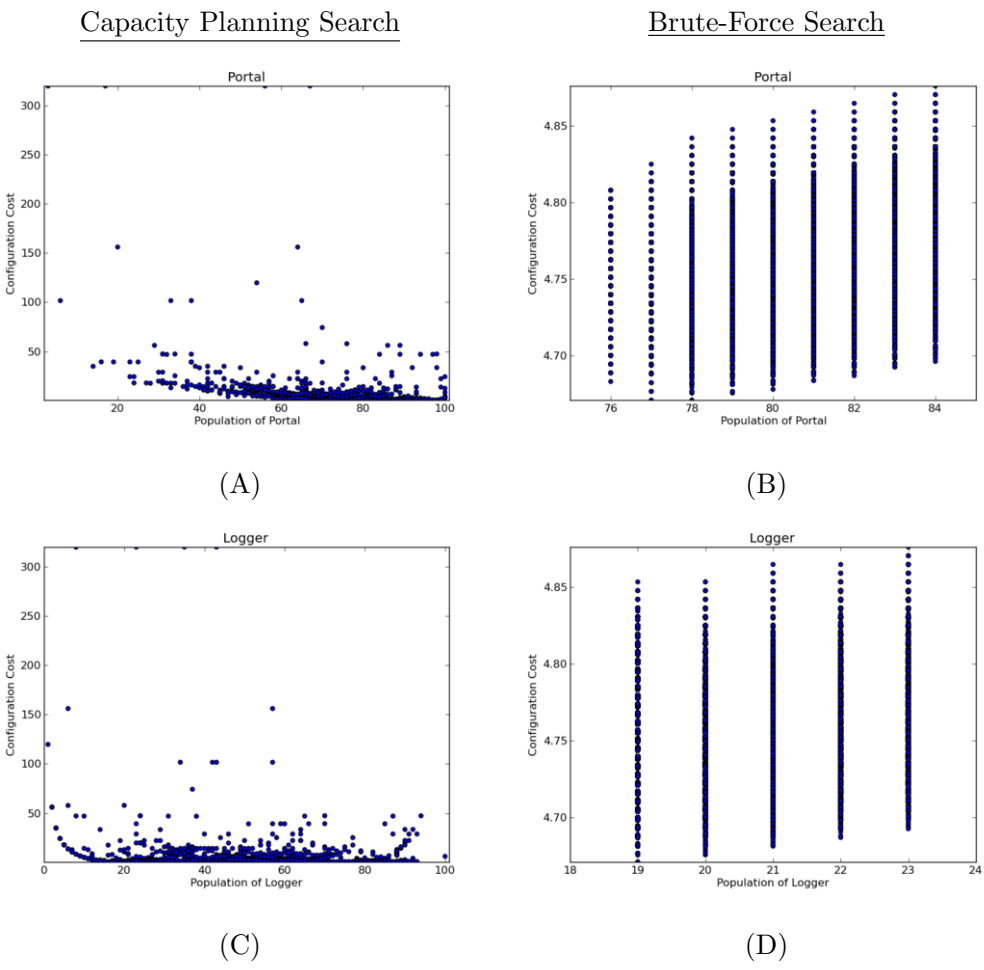

Fig. 2. Scatter plots showing the results for a selection of the configurable server components. The left-hand graphs depict the results for the capacity planning search whilst the right hand graphs depict the results for the brute-force search. The x-axis ranges are much smaller for the brute-force search since it is infeasible to evaluate all configurations when the range of possible values is large.

value is an approximation to a non-linear cost function. We think this is a good trade-off of complexity, easy of use and power of expression. However, we continue to investigate other possibilities.

Finally throughout this paper we have assumed that the modeller can either make a good guess to the level of expected demand or is prepared to over-estimate it. A further possibility is to perform multiple capacity planning searches assuming different levels of demand.

We could perform multiple capacity planning searches for different levels of demand automatically. Furthermore we may see adaptability to different levels of demand as a particularly good thing to have. For example some services can operate at different levels, in the most obvious case by simply turning servers on or off. Currently, whilst we may find a configuration which is particularly good for a particular level of demand it may not be very adaptable. Hence we continue to

investigate ways in which we may reward configurations that are adaptable.

In the meantime we provide methods for the modeller to examine some of the configurations that have been found mid-search, but perhaps did not have the globally best cost, because adaptability is not accounted for in the cost function. This provides a further reason that it is particularly useful for the modeller to be able to examine elements of the search rather than simply the best configuration found.

5 Conclusion

When modelling service based systems such as the system modelled in our case study the modeller is unlikely to have great control over the level of demand. Therefore the system designer must be sure to provision enough service to satisfy a realistic level of demand.

Most realistic levels of demand can be satisified with enough service provision, but generally there is some significant cost to providing that level of service. If not then one need do little modelling but simply provision plenty of service component.

Assuming that there is some significant cost we would like to know how best to provide the required level of service. Even further we may not know the level of service we demand but we have some idea of how to trade-off the level of performance against the cost of the provision.

However, knowing this is not enough for many kinds of services. These are services in which there are more two kinds of components that need to be deployed to provide the whole service. In these kinds of scenarios there are simply too many plausible configurations of the service to try them all. Furthermore it is rarely obvious what the most efficient configuration is, or even how to improve on the current one.

Hence an automatic search through the configuration space can provide excellent insight for the modeller. We are of the opinion that not only the end result of such a search but many of the configurations and their associated costs found mid-search may be of interest to the modeller.

Performing such a search is a non-trivial task. A user-friendly GUI based tool which not only performs the search itself but guides the user through the configuration of the search is a significant help to the modeller. We have presented such a software tool in this paper.

The trade-off is that the developers of such a tool must consider the ways in which the modeller may wish to evaluate the efficacy of a particular model configuration. We think that so far we have a powerfully expressive method of configuration but we continue to investigate methods to be more expressive as well as more intuitive.

A problem with a brute-force search as opposed to a search heuristic, is there are so many configurations to solve we must reduce the available flexibility meaning that the modeller must already have significant insight into their model. Capacity planning can either be used on its own, or to find a good set of ranges in which to perform an exhaustive search. Indeed a good workflow is to use a driven PSO search to find an optimal area of the search space, and then to use a brute-force

search to exhastively search the local area.

Finally we wish to claim that capacity planning, or more generally a heuristic search, is a useful addition to any modelling software. It is difficult to provide the correct interface, but this is ultimately worth the effort. The capacity planning extension to the PEPA Eclipse Plug-in project [16] is now available as of October 2014.

References

[1] Bäck, T. and H.-P. Schwefel, *An overview of evolutionary algorithms for parameter optimization*, Evol. Comput. **1** (1993), pp. 1–23.
URL http://dx.doi.org/10.1162/evco.1993.1.1.1

[2] Clark, A. and S. Gilmore, *State-aware performance analysis with eXtended Stochastic Probes*, in: N. Thomas and C. Juiz, editors, *Proceedings of the 5th European Performance Engineering Workshop (EPEW 2008)*, LNCS **5261** (2008), pp. 125–140.

[3] Clark, A. and S. Gilmore, *Transformations in PEPA Models and Stochastic Probe Placement*, in: K. Djemame, editor, *Proceedings of the Twenty-Fifth UK Performance Engineering Workshop*, Leeds University, 2009, pp. 1–16.

[4] Deb, K. and D. Kalyanmoy, "Multi-Objective Optimization Using Evolutionary Algorithms," John Wiley & Sons, Inc., New York, NY, USA, 2001.

[5] Dingle, N. J. and W. J. Knottenbelt, *Automated Customer-Centric Performance Analysis of Generalised Stochastic Petri Nets Using Tagged Tokens*, in: *Third International Workshop on Practical Applications of Stochastic Modelling (PASM'08), Palma de Mallorca, Spain*, 2008.
URL http://pubs.doc.ic.ac.uk/pasm08-tagged/

[6] Elgner, J., S. Gnesi, N. Koch and P. Mayer, *Introduction to the sensoria case studies*, in: M. Wirsing and M. Hölzl, editors, *Rigorous Software Engineering for Service-oriented Systems*, Springer-Verlag, Berlin, Heidelberg, 2011 pp. 26–34.
URL http://dl.acm.org/citation.cfm?id=2043021.2043025

[7] Hillston, J., *The nature of synchronisation*, in: U. Herzog and M. Rettelbach, editors, *Proceedings of the Second International Workshop on Process Algebras and Performance Modelling*, Erlangen, 1994, pp. 51–70.

[8] Hillston, J., "A Compositional Approach to Performance Modelling," Cambridge University Press, 1996.

[9] Hillston, J., *Fluid flow approximation of PEPA models*, in: *Proceedings of the Second International Conference on the Quantitative Evaluation of Systems* (2005), pp. 33–43.

[10] Hillston, J., *Process algebras for quantitative analysis*, in: *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)* (2005), pp. 239–248.

[11] Hillston, J., *Tuning systems: From composition to performance*, The Computer Journal **48** (2005), pp. 385–400, the Needham Lecture paper.

[12] Hillston, J., M. Tribastone and S. Gilmore, *Stochastic process algebras: From individuals to populations*, Comput. J. **55** (2012), pp. 866–881.

[13] Hillston, J. and C. D. Williams, *Capacity planning for PEPA models*, in: A. Horvath and K. Wolter, editors, *Proceedings of the 11th European Performance Engineering Workshop (EPEW 2014)*, LNCS **8721** (2014), to appear in.

[14] Little, J. D. C., *A proof of the queueing formula* $l = \lambda w$, Operations Research **9** (1961), pp. 380–387.

[15] Simon, D., "Evolutionary Optimization Algorithms," 2013.

[16] Tribastone, M., *The PEPA Plug-in Project*, in: M. Harchol-Balter, M. Kwiatkowska and M. Telek, editors, *Proceedings of the 4th International Conference on the Quantitative Evaluation of SysTems (QEST)* (2007), pp. 53–54.

[17] Tribastone, M., "Scalable Analysis of Stochastic Process Algebra Models," Ph.D. thesis, School of Informatics, The University of Edinburgh (2010).
URL https://dl.dropboxusercontent.com/u/13100903/papers/phd-thesis.pdf